(2)

AVF Control Number: AVF-VSR-346.0390
89-09-20-ICC

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 891212W1.10203
Irvine Compiler Corporation
ICC Ada, 6.0.0
HP 9000 Model 350 Host and
HP 64000 UX with 68020 Emulation Pod Target

Completion of On-Site Testing:
12 December 1989

Prepared By:
Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

AD-A221 653

OFFICE OF THE DIRECTOR OF
DEFENSE RESEARCH AND ENGINEERING

WASHINGTON, DC 20301

19 APR 199

(R&AT)

MEMORANDUM FOR Director, Directorate of Database Services,
Defense Logistics Agency

SUBJECT: Technology Screening of Unclassified/Unlimited Reports

Your letter of 2 February 1990 to the Commander, Air Force Systems Command, Air Force Aeronautical Laboratory, Wright-Patterson Air Force Base stated that the Ada Validation Summary report for Meridian Software Systems, Inc. contained technical data that should be denied public disclosure according to DoD Directive 5230.25

We do not agree with this opinion that the contents of this particular Ada Validation Summary Report or the contents of the several hundred of such reports produced each year to document the conformity testing results of Ada compilers. Ada is not used exclusively for military applications. The language is an ANSI Military Standard, a Federal Information Processing Standard, and an International Standards Organization standard. Compilers are tested for conformity to the standard as the basis for obtaining an Ada Joint Program Office certificate of conformity. The results of this testing are documented in a standard form in all Ada Validation Summary Reports which the compiler vendor agrees to make public as part of his contract with the testing facility.

On 18 December 1985, the Commerce Department issued Part 379 Technical Data of the Export Administration specifically listing Ada Programming Support Environments (including compilers) as items controlled by the Commerce Department. The AJPO complies with Department of Commerce export control regulations. When Defense Technical Information Center receives an Ada Validation Summary Report, which may be produced by any of the five U.S. and European Ada Validation Facilities, the content should be made available to the public.

If you have any further questions, please feel free to contact the undersigned at (202) 694-0209.

John P. Solomond

John P. Solomond
Director
Ada Joint Program Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Ada Compiler Validation Summary Report: Irvine Compiler Corporation, ICC Ada, 6.0.0, HP 9000 Model 350 (Host) to HP 64000 UX with 68020 (Target), 891212W1.10203

5. TYPE OF REPORT & PERIOD COVERED
12 Dec. 1989 to 12 Dec. 1990

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Wright-Patterson AFB
Dayton, OH, USA

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS
Wright-Patterson AFB
Dayton, OH, USA

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
Wright-Patterson AFB
Dayton, OH, USA

15. SECURITY CLASS (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
Irvine Compiler Corporation, ICC Ada, 6.0.0, Wright-Patterson AFB, HP 9000 Model 350 under HP-UX, Release 6.2 (Host) to HP 64000 UX with 68020 Emulation Pode (bare machine) (Target), ACVC 1.10.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Compiler Validation Summary Report:

Compiler Name: ICC Ada, 6.0.0

Certificate Number: 891212W1.10203

Host: HP 9000 Model 350 under
HP-UX, Release 6.2

Target: HP 64000 UX with 68020 Emulation Pod
bare machine

Testing Completed 12 December 1989 Using ACVC 1.10

Customer Agreement Number: 89-09-20-ICC

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report ~~(VSR)~~ describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability, ~~(ACVC)~~. An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

(KR)

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 12 December 1989 at Irvine CA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C.#552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures, Version 2.0, Ada Joint Program Office, May 1989.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation of legal Ada programs with certain language constructs which cannot be verified at compile time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: ICC Ada, 6.0.0

ACVC Version: 1.10

Certificate Number: 891212W1.10203

Host Computer:

Machine:	HP 9000 Model 350
Operating System:	HP-UX Release 6.2
Memory Size:	16 Megabytes

Target Computer:

Machine:	HP 64000 UX with 68020 Emulation Pod
Operating System:	bare machine
Memory Size:	1 Megabyte

Communications Network: HP-IB link

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined types TINY_INTEGER and SHORT_INTEGER in package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

(4) NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is not gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round away from zero. (See tests C46012A..Z (26 tests).)

(2) The method used for rounding to longest integer is round away from zero. (See tests C46012A..Z (26 tests).)

(3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

(2) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components with each component being a null array. (See test C36202A.)

(3) CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components with each component being a null array. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises CONSTRAINT_ERROR when the array objects are declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the array objects are declared. (See test C52104Y.)

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises CONSTRAINT_ERROR when the array type is declared. (See test E52103Y.)

(7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(8) In assigning two-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, index subtype checks are made as choices are evaluated. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised before all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

(1) The pragma INLINE is not supported for functions or procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

(1) The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

(2) The package DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

(3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E (2 tests), CE2102N, and CE2102P.)

(4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

(5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

(6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

(7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

(8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

(9) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

(10) Temporary sequential files are given names and deleted when closed. (See test CE2108A.)

(11) Temporary direct files are given names and deleted when closed. (See test CE2108C.)

(12) Temporary text files are given names and deleted when closed. (See test CE3112A.)

(13) More than one internal file can be associated with each external file for sequential files when writing or reading. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

(14) More than one internal file can be associated with each external file for direct files when writing or reading. (See tests CE2107F..H (3 tests), CE2110D, and CE2111H.)

(15) More than one internal file can be associated with each external file for text files when writing or reading. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 444 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 5 tests were required to successfully demonstrate the test objective.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	125	1131	1893	17	19	44	3229
Inapplicable	4	7	422	0	9	2	444
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	568	545	244	172	99	159	332	131	36	252	211	282	3229
Inappl	14	81	135	4	0	0	7	0	6	0	0	158	39	444
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84M	CD2A84N	CD2B15C	CD2D11B	CD5007B	CD50110
ED7004B	ED7005C	ED7005D	ED7006C	ED7006D	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 444 tests were inapplicable for the reasons indicated:

a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)

C45241L..Y (14 tests) C45321L..Y (14 tests)
C45421L..Y (14 tests) C45521L..Z (15 tests)
C45524L..Z (15 tests) C45621L..Z (15 tests)
C45641L..Y (14 tests) C46012L..Z (15 tests)

b. The following 12 tests are not applicable because this implementation does not support an enumeration type with a representation clause used as a generic actual parameter:

C35502J C35502N C35507J C35507N C35508J
C35508N CD2A21E CD2A22I CD2A22J CD2A23E
CD2A24I CD2A24J

c. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

d. C35702B and B86001U are not applicable because this implementation supports no predefined type LONG_FLOAT.

e. The following 30 tests are not applicable because this implementation does not support an access type length clause:

A39005C C87B62B CD1009J CD1009R..S (2)
CD1C03C CD2A83A..C (3) CD2A83E..F (2) CD2A84B..I (8)
CD2A84K..L (2) ED2A86A CD2B11B..G (6) CD2B15B
CD2B16A

f. The following 21 tests are not applicable because this implementation does not support a SMALL length clause for fixed point types:

A39005E C87B62C CD1009L CD1C03F
CD1C04C CD2A53A..E (5) CD2A54A..D (4) CD2A54G..J (4)
ED2A56A CD2D11A CD2D13A

g. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C C45304C C45502C C45503C C45504C
C45504F C45611C C45613C C45614C C45631C
C45632C B52004D C55B07A B55B09C B86001W
CD7101F

h. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 48.

i. C55B16A is not applicable because this implementation does not support a negative value for an enumeration literal in an enumeration representation clause.

j. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

k. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

l. LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F are not applicable because this implementation does not support pragma INLINE.

m. CD1009C, CD2A41A..B (2 tests), CD2A41E, and CD2A42A..J (10 tests) are not applicable because this implementation does not support a size clause on a floating point type less than the size of the smallest predefined type.

n. CD1C03H, CD1C04E, ED1D04A, and CD4051B are not applicable because this implementation does not support out of order fields for record representation clauses.

o. CD2A32I..J (2 tests) are not applicable because this implementation does not support an integer type with a representation clause used as a generic actual parameter.

p. The following 14 tests are not applicable because this implementation does not support a fixed point type with a representation clause used as a generic actual parameter:

CD2A51E	CD2A52I	CD3014A..B (2)	CD3014D..E (2)
CD3015A..B (2)	CD3015D..E (2)	CD3015G	CD3015I..J (2)
CD3015L			

q. CD2A52G..H (2 tests), and CD2A52J are not applicable because this implementation does not support a fixed point type with a size clause less than the smallest predefined fixed point type.

r. The following 32 tests are not applicable because this implementation does not support crossing 32-bit packing unit boundaries in representation clauses:

CD2A61A..D (4)	CD2A61F	CD2A61K..L (2)	CD2A62A..C (3)
CD2A64C..D (2)	CD2A65A..D (4)	CD2A71A..D (4)	CD2A72A..D (4)
CD2A74A..D (4)	CD2A75A..D (4)		

s. CD2A61H..J (3 tests) are not applicable because this implementation does not support representation clauses with implicit modification of storage requirements for component types.

t. CD2A64A..B (2 tests) are not applicable because this implementation does not support length clauses on derived types.

u. CD4051D is not applicable because this implementation does not support a representation clause for a field in an unconstrained variant record.

v. The following 29 tests are not applicable because this implementation does not support an address clause for constant objects:

CD5011B	CD5011D	CD5011F	CD5011H	CD5011L
CD5011N	CD5011R	CD5012C	CD5012D	CD5012G
CD5012H	CD5012L	CD5013B	CD5013D	CD5013F
CD5013H	CD5013L	CD5013N	CD5013R	CD5014B
CD5014D	CD5014F	CD5014H	CD5014J	CD5014L
CD5014N	CD5014R	CD5014U	CD5014W	

w. AE2101C and EE2201D..E (2 tests) use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

x. AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

y. CE2102C, CE2102H, CE2103A..B (2 tests), CE3102B, CE3107A were ruled not applicable by the AVO because this implementation uses a "Virtual I/O" package to simulate a disk file. The VIRTUAL_IO package does not have illegal filenames which cause these tests to report failed or raise an unhandled exception.

z. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

aa. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

ab. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

ac. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

ad. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

ae. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

af. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

ag. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

ah. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

ai. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

aj. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

ak. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

al. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

am. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

an. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

ao. CE2108B, CE2108D, CE2108F, CE2108H, CE3112B, CE3112D were ruled not applicable by the AVO because this implemtation uses a file-system simulated in RAM, so it is not possible for files to exist between the execution of ACVC tests.

ap. CE3102E is inapplicable because this implementation supports CREATE with IN_FILE mode for text files.

aq. CE3102F is inapplicable because this implementation supports RESET for text files.

ar. CE3102G is inapplicable because this implementation supports deletion of an external file for text files.

as. CE3102I is inapplicable because this implementation supports CREATE with OUT_FILE mode for text files.

at. CE3102J is inapplicable because this implementation supports OPEN with IN_FILE mode for text files.

au. CE3102K is inapplicable because this implementation supports OPEN with OUT_FILE mode for text files.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 5 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B24009A B49003A B49005A B59001E

The following modifications were made to compensate for legitimate implementation behavior:

a. In test CC1223A the expression "2**T'MANTISSA - 1" on line 262 causes an unexpected exception to be raised. The test was modified, as approved by the AVO, to use a temporary variable to re-order the association of operands in the expression "(2**(T'MANTISSA-1)-1 + 2**(T'MANTISSA-1))". The following section of code results:

```
DECLARE                                                    -- NEW
   TEMP: INTEGER := 2**(T'MANTISSA-1)-1;                   -- NEW
BEGIN                                                      -- NEW
   IF T'LARGE /=
        (TEMP + 2**(T'MANTISSA-1)) * T (T'SMALL) THEN -- NEW
        FAILED ( "INCORRECT VALUE FOR " &
                  STR & "'LARGE" );
   END IF;
END;                                                       -- NEW
```

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the ICC Ada, 6.0.0 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the ICC Ada, 6.0.0 compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	HP 9000 Model 350
Host operating system:	HP-UX, Release 6.2
Target computer:	HP 64000 UX with 68020 Emulation Pod
Target operating system:	Bare Machine
Compiler:	ICC Ada, 6.0.0

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded onto the host computer over Ethernet using NFS from a remote host with a directly connected 9-track tape drive.

After the test files were loaded to disk, the full set of tests was compiled and linked on the HP 9000 Model 350. All executable tests were downloaded via an HP-IB link to the HP 64000 UX emulator with a 68020 emulation pod and executed in the emulator. Each test's output was captured and sent to a disk file on the host. Results were printed by a laser printer on a remote server connected by Ethernet.

The compiler was tested using command scripts provided by Irvine Compiler Corporation and reviewed by the validation team. The compiler was tested using all the following option settings. See Appendix E for a complete listing of the compiler options for this implementation. The following list of compiler options includes those options which were invoked by default:

-lib	Specify which compilation library to use.
-stack_check	Enable stack overflow checks.
-numeric_check	Enable arithmetic overflow checks.
-elaboration_check	Enable elaboration checks.
-advise	Suppress advisory warnings.
-quiet	Suppress compiler banners.
-link	Link the provided program.

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Irvine CA and was completed on 12 December 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Irvine Compiler Corporation has submitted the following Declaration of Conformance concerning the ICC Ada, 6.0.0 compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: Irvine Compiler Corporation
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name: ICC Ada Release: 6.0.0
Host Architecture ISA:HP 9000 Model 350 OS&VER#: HP-UX Release 6.2
Target Architecture ISA: HP 64000 UX OS&VER#: bare machine
with 68020 Emulation Pod

Implementor's Declaration

I, the undersigned, representing Irvine Compiler Corporation, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Irvine Compiler Corporation is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

Dan Eilers Date: 10/16/89
Irvine Compiler Corporation
Dan Eilers, President

Owner's Declaration

I, the undersigned, representing Irvine Compiler Corporation, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A as measured by ACVC 1.10.

Dan Eilers Date: 10/16/89
Irvine Compiler Corporation
Dan Eilers, President

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the ICC Ada, 6.0.0 compiler, as described in this Appendix, are provided by Irvine Compiler Corporation. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

   type INTEGER is range -2147483648 .. 2147483647;
   type SHORT_INTEGER is range -32768 .. 32767;
   type TINY_INTEGER is range -128 .. 127;

   type FLOAT is digits 15
        range -1.2355820928895E+307 .. 1.2355820928895E+307;

   type DURATION is delta 2.0**(-12) range -524287.0 .. 524287.0;

   ...

end STANDARD;
```

ICC Ada Version 6.0.0
HP-9000/350 to HP-64000/68020 pod
HP-UX, Release 6.2

Irvine Compiler Corporation
18021 Sky Park Circle, Suite L
Irvine, CA 92714
(714) 250-1366

December 5, 1989

1 ICC Ada Implementation

The Ada language definition leaves implementation of certain features to the language implementor. This appendix describes the implementation-dependent characteristics of ICC Ada.

2 Pragmas

The following predefined pragmas are implemented in ICC Ada as described by the Ada Reference Manual:

Elaborate This pragma allows the user to modify the elaboration order of compilation units.

List This pragma enables or disables writing to the output list file.

Pack Packing on arrays is implemented to the bit level. The current implementation packs records to the byte level. Slices of packed arrays are not implemented, except boolean arrays.

Page This pragma ejects a new page in the output list file (if enabled).

Priority This pragma sets the priority of a task or main program. The range of the subtype **priority** is 0..254.

The following predefined pragmas have been extended by ICC:

Interface This pragma is allowed to designate variables in addition to subprograms. It is also allowed to have an optional third parameter which is a string designating the name for the linker to use to reference the variable or subprogram. The third parameter has the same effect as pragma interface_name.

Suppress In addition to suppressing the standard checks, ICC also permits suppressing the following:

Exception_info Suppressing **exception_info** improves run-time performance by reducing the amount of information maintained for messages that appear when exceptions are propagated out of the main program or any task.

All_checks Suppressing **all_checks** suppresses all the standard checks as well as **exception_info**.

The following predefined pragmas are currently not implemented by ICC:

Controlled, Inline, Memory_size, Optimize

Shared, Storage_unit, System_name

The following additional pragmas have been defined by ICC:

Compress This pragma reduces the storage required for discrete subtypes in structures (arrays and records). Its single argument is the name of a discrete subtype. It specifies that the subtype should be represented as compactly as possible (regardless of the representation of the subtype's base type) when the subtype is used in a structured type. The storage requirement for variables and parameters is not affected. This pragma must appear prior to any reference to the named subtype.

Export This pragma is a complement to the predefined pragma **interface**. It enables subprograms written in Ada to be called from other languages. It takes 2 or 3 arguments. The first is the language to be called from, the second is the subprogram name, and the third is an optional string designating the actual subprogram name to be used by the linker. This pragma must appear prior to the body of the designated subprogram.

Interface_Name This pragma takes a variable or subprogram name and a string to be used by the linker to reference the variable or subprogram. It has the same effect as the optional third parameter to pragma **interface**.

No_zero The single parameter to **no_zero** is the name of a record type. If the named record type has "holes" between fields that are normally initialized with zeroes, this pragma will suppress the clearing of the holes. If the named record type has no "holes" this pragma has no effect. When zeroing is disabled, comparisons (equality and non-equality) of the named type are disallowed. The use of this pragma can significantly reduce initialization time for record objects. The ICC Command Interpreter also has a flag **-no_zero** which has the effect of implicitly applying pragma **no_zero** to all record types declared in the file.

Put, Put_line These pragmas take any number of arguments and write their value to standard output at compile time when encountered by the compiler. The arguments may be expressions of any string, enumeration, or integer type, whose value is known at compile time. Pragma **put_line** adds a carriage return after printing all of its arguments. These pragmas are often useful in conjunction with conditional compilation. They may appear anywhere a pragma is allowed.

Unsigned_Literal This pragma applied to any 32-bit signed integer type affects the interpretation of literals for such a type. Specifically literals between 2**31 and 2**32 are represented as if the type was unsigned. Operations on the type are unaffected. Note however that (with checking suppressed), signed addition, subtraction, and multiplication are identical to the corresponding unsigned operations. However, division and inequalities are different and should be used with utmost caution. This pragma is used for type **address** in package **system**.

3 Preprocessor Directives

ICC Ada incorporates an integrated preprocessor whose directives begin with the keyword **pragma**. They are as follows:

If, Elsif, Else, End These preprocessor directives provide a conditional compilation mechanism. The directives **if** and **elsif** take a boolean static expression as their single argument. If the expression evaluates to **FALSE** then all text up to the next **end**, **elsif** or **else** directive is ignored. Otherwise, the text is compiled normally. The usage of these directives is identical to that of the similar Ada constructs. These directives may appear anywhere pragmas are allowed and can be nested to any depth.

Include This preprocessor directive provides a compile-time source file inclusion mechanism. It is integrated with the library management system, and the automatic recompilation facilities.

The results of the preprocessor pass, with the preprocessor directives deleted and the appropriate source code included, may be output to a file at compile-time.

4 Attributes

ICC Ada implements all of the predefined attributes, including the Representation Attributes described in section 13.7 of the Ada RM.
Limitations of the predefined attributes are:

Address This attribute cannot be used with a statement label or a task entry.

Storage_size Since ICC Ada does not allocate dynamic storage using "collections", this attribute always returns a constant value.

The implementation defined attributes for ICC Ada are:

Version, System, Target, CG_mode These attributes are used by ICC for conditional compilation. The prefix must be a discrete type. The values returned vary depending on the target architecture and operating system.

5 Input/Output Facilities

5.1

The implementation dependent specifications from TEXT_IO and DIRECT_IO are:

```
type COUNT is range 0 .. integer'last;
subtype FIELD is INTEGER range 0 .. integer'last;
```

5.2 FORM Parameter

ICC Ada implements the FORM parameter to the procedures OPEN and CREATE in DIRECT_IO, SEQUENTIAL_IO, and TEXT_IO to perform a variety of ancillary functions. The FORM parameter is a string literal containing parameters in the style of named parameter notation. In general the FORM parameter has the following format:

"$field_1$ => $value_1$ [, $field_n$ => $value_n$]"

where $field_i$ => $value_i$ can be

OPTION	=>	NORMAL
OPTION	=>	APPEND
PAGE_MARKERS	=>	TRUE
PAGE_MARKERS	=>	FALSE
READ_INCOMPLETE	=>	TRUE
READ_INCOMPLETE	=>	FALSE
MASK	=>	<9 character protection mask>

Each *field* is separated from its *value* with a "=>" and each *field/value* pair is separated by a comma. Spaces may be added anywhere between tokens and upper-case/lower-case is insignificant. For example:

```
create( f, out_file, "list.data",
    "option => append, PAGE_MARKERS => FALSE, Mask => rwxrwx---");
```

The interpretation of the fields and their values is presented below.

OPTION Files may be OPENed for appendage. This causes data to be appended directly onto the end of an existing file. The default is NORMAL which overwrites existing data. This field applies to OPEN in all three standard I/O packages. It has no effect if applied to procedure CREATE.

PAGE_MARKERS If FALSE then all TEXT_IO routines dealing with page terminators are disabled. They can be called, however they will not have any effect. In addition the page terminator character (^L) is allowed to be read with GET and GET_LINE. The default is TRUE which leaves page terminators active. Disabling page terminators is particularly useful when using TEXT_IO with an interactive device. For output files, disabling page terminators will suppress the page terminator character that is normally written at the end of the file.

READ_INCOMPLETE This field applies only to DIRECT_IO and SEQUENTIAL_IO and dictates what will be done with reads of incomplete records. Normally, if a READ is attempted and there is not enough data in the file for a complete record, then END_ERROR or DATA_ERROR will be raised. By setting READ_INCOMPLETE to TRUE, an incomplete record will be read successfully and the remaining bytes in the record will be zeroed. Attempting a read after the last incomplete record will raise END_ERROR. The SIZE function will reflect the fact that there is one more record when the last record is incomplete and READ_INCOMPLETE is TRUE.

MASK Set a protection mask to control access to a file. The mask is a standard nine character string notation used by Unix. The letters cannot be rearranged or deleted so that the string is always exactly nine characters long. This applies to CREATE in all three standard I/O packages. The default is determined at runtime by the user's environment settings.

The letters in the mask are used to define the Read, Write and eXecute permissions for the User, Group and World respectively. Wherever the appropriate letter exists, the corresponding privilege is granted. If a "-" is used instead, then that privilege is denied. For example if mask were set to "rw-rw----" then read and write privilege is granted to the file owner and his/her group, but no world rights are given.

If a syntax error is encountered within the FORM parameter then the exception USE_ERROR is raised at the OPEN or CREATE call. Also, the standard function TEXT_IO.FORM returns the current setting of the form fields, including default values, as a single string.

6 Package System

Package SYSTEM is:

```
package SYSTEM is

   type name is (m68000, m68010, m68020, m68030);

   -- Language Defined Constants

   system_name : constant name := m68020;       -- HP64000 with 68020 pod.
   storage_unit: constant := 8;                 -- Storage unit size in bits.
   memory_size : constant := 1_024 * 1_024;     -- 1 Megabyte.
   min_int     : constant := -2**31;
   max_int     : constant := 2**31-1;
   max_digits  : constant := 15;
   max_mantissa: constant := 31;
   fine_delta  : constant := 2.0**(-31);
   tick        : constant := 1.0/60.0;

   type address is range min_int..max_int;      -- Signed 32 bit range.
   subtype priority is integer range 0..254;    -- 0 is default priority.

   -- Constants for the HEAPS package

   bits_per_bmu : constant := 8;                -- Bits per basic machine unit.
   max_alignment: constant := 4;                -- Maximum alignment required.
   min_mem_block: constant := 1024;             -- Minimum chunk request size.

   -- Constants for the HOST package

   host_clock_resolution: constant := 1;        -- 1 microsecond.
   base_date_correction : constant := 25_202;   -- Unix base date is 1/1/1970.
```

```
   pragma unsigned_literal(address);          -- Allow unsigned literals.

   null_address: constant address := 0;       -- Value of type ADDRESS
                                              -- equal to NULL.

   pragma put_line("Target: ", system_name);
end SYSTEM;
```

7 Limits

Most data structures held within the ICC Ada compiler are dynamically allocated, and hence have no inherent limit (other than available memory). Some limitations are:

The maximum input line length is 254 characters.

The maximum number of tasks abortable by a single abort statement is 64.

Include files can be nested to a depth of 3.

The number of packages, subprograms, tasks, variables, aggregates, types or labels which can appear in a compilation unit is unlimited.

The number of compilation units which can appear in one file is unlimited.

The number of statements per subprogram or block is unlimited.

Packages, tasks, subprograms and blocks can be nested to any depth.

There is no maximum number of compilation units per library, nor any maximum number of libraries per library system.

8 Numeric Types

ICC Ada supports three predefined integer types:

TINY_INTEGER	-128..127	8 bits
SHORT_INTEGER	-32768..32767	16 bits
INTEGER	-2147483648..2147483647	32 bits

In addition, unsigned TINY and SHORT integer types can be defined by the user via the SIZE representation clause. Storage requirements for types can be reduced by using pragma **pack** and for subtypes by using the ICC pragma **compress**.

Type **float** is available.

Attribute	**FLOAT** value
size	64 bits
digits	15
first	$-1.12355820928895E+307$
last	$+1.12355820928895E+307$

9 Tasks

The type DURATION is defined with the following characteristics:

Attribute	**DURATION** value
delta	$2.44140625E-04$ sec
small	$2.44140625E-04$ sec
first	-524287.0 sec
last	524287.0 sec

The subtype SYSTEM.PRIORITY as defined provides the following range:

Attribute	**PRIORITY** value
first	0
last	254

Higher numbers correspond to higher priorities. If no priority is specified for a task, PRIORITY'FIRST is assigned during task creation.

10 Representation Clauses

10.1 Type Representation Clauses

Many of the type representation clauses have been implemented. Their implementation is detailed in this section.

NOTE: Type conversions that involve representation changes are *not* implemented. Representation clauses (and the **PACK** pragma) should be applied to non-derived types only.

10.1.1 Length Clauses

The amount of storage to be associated with an entity is specified by means of a length clause. The following is a list of length clauses and their implementation status:

- The 'SIZE length clause is implemented, with restrictions. When applied to integer range types this representation clause can be used to reduce storage requirements and also to define types with unsigned representation. It is currently *not* possible to declare an unsigned 32-bit type in ICC

Ada. Enumeration, integer, or fixed-point types with length clauses are not allowed as generic parameters. Length clauses *are* allowed for float and fixed point types, however the storage requirements for these types *cannot* be reduced below the smallest applicable predefined type available.

- The 'STORAGE_SIZE length clause for task types is implemented. The size specified is used to allocate both the task's Task Information Block (TIB) *and* its stack.
- The 'STORAGE_SIZE length clause for access types is not implemented.
- The 'SMALL length clause for fixed point types is not implemented.

10.1.2 Enumeration Representation Clauses

Enumeration representation clauses are implemented. However, enumerations with representation clauses are not allowed as generic parameters. The use of enumeration representation clauses can greatly increase the overhead associated with their reference. In particular, **FOR** loops on such enumerations are very expensive. The values associated with enumeration literals in an enumeration representation clause must be non-negative. Representation clauses which define the default representation (i.e. The first element is ordinal 0, the second 1, the third 2, etc.) are detected and cause no additional overhead.

10.1.3 Record Representation Clauses

Record representation clauses are implemented for simple records. Records containing discriminants and dynamic arrays may not be organized as expected because of unexpected changes of representation. There are no implementation generated names that can be used in record representation clauses. Out of order fields or offsets for record representation clause are not implemented. Representation clauses for a field not in a constrained record variant are not implemented.

10.2 Storage Algorithms

This section describes the default and packed representation of record and array types.

10.2.1 Default Memory Allocation

ICC Ada allocates space for record fields in the order in which they are declared. If the record is not packed, then the following rules are used to allocate offsets for each field:

- For objects smaller than a byte, the size of the object is made one byte.

- Each field is assigned a *byte* offset according to the alignment requirements of the target hardware. For the 68020 processor, the compiler's default alignment rules require that byte sized objects be aligned to any byte address, word sized objects be aligned on 2-byte boundaries and long sized objects be aligned on 4-byte boundaries. Short_float and float objects are aligned on 4-byte boundaries.

The default algorithm assigns a byte address to every field in the record. Although this may waste memory storage, it provides for fast access to each field within the record sinnce bit extraction operations are not needed.

Example: For the following code segment, f1 is allocated one byte although it only requires 6 bits. Likewise, f3 is allocated one byte although it only requires 7 bits. F2 and f4 are allocated 2 bytes each to round them up to the nearest *natural* integer size. Furthermore, since the processor requires word-sized objects to be aligned on even byte boundaries, the four fields end up using 8 bytes, leaving two odd addressed bytes unused. See figure 1.

```
type int6 is range 0..2**6-1;
type int7 is range 0..2**7-1;
type int9 is range 0..2**9-1;
type int10 is range 0..2**10-1;

for int6'size use 6;            -- INT6  is unsigned  6-bit integer
for int7'size use 7;            -- INT7  is unsigned  7-bit integer
for int9'size use 9;            -- INT9  is unsigned  9-bit integer
for int10'size use 10;          -- INT10 is unsigned 10-bit integer

type sample_rec1 is
  record
    f1 : int6;          -- 6  bit field
    f2 : int10;         -- 10 bit field
    f3 : int7;          -- 7  bit field
    f4 : int9;          -- 9  bit field
  end record;
```

10.2.2 Packed Memory Allocation

When pragma *Pack* or a representation clause is used, the following rules are ***always*** used to allocate memory:

- Structure types (arrays and records) are packed into 32-bit *packing units*. The compiler packs each field of the type at the bit level into groups of 32-bits.

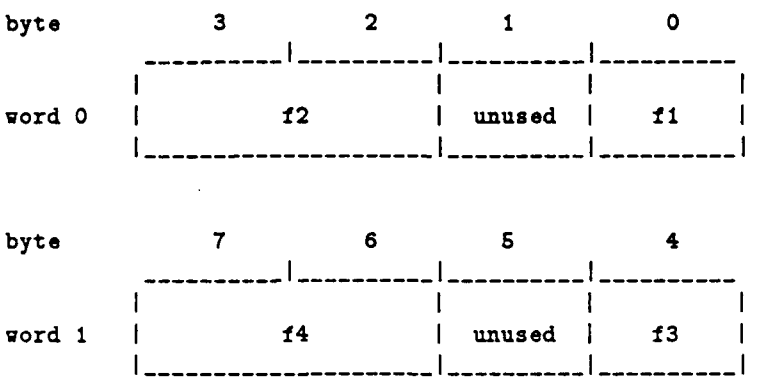

Figure 1. Memory Allocation for an Unpacked Record Structure

- Byte boundaries can be crossed (within packing units) but word boundaries cannot be crossed.
- When pragma pack or a length clause does not reduce the amount of memory used, then the representation is not modified.
- Structure objects always start on a byte boundary.
- The objects are aligned in memory according to the hardware alignment requirements.
- Neither pragma pack nor a representation clause will automatically modify the storage requirements or allocation rules of a component type (in the case of arrays or records) in an effort to reduce storage. The storage requirements for each component type must be reduced explicitly.

If the record **sample_rec1** in figure 1 is packed, then the entire record would fit in one word as is illustrated in figure 2.

Word Boundaries Not Crossed. When packed objects are allocated space in memory, they are packed to 32 bit packing units. Since each field is being packed into these word-sized units, byte boundaries within the word are ignored, but fields are not allowed to cross between word boundaries.

Example: If a record is declared with four fields consisting of 6 bits, 10 bits, 8 bits, and 10 bits wide, the first three fields (totalling 24 bits) are allocated contiguously in the first word (packing unit), leaving the remaining 8 bits in the

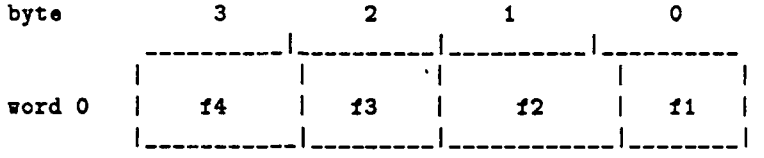

Figure 2. Memory Allocation for a Packed Record Structure

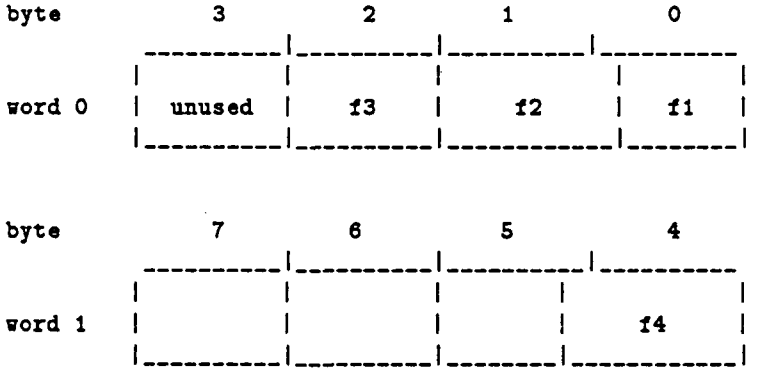

Figure 3. Memory Allocation for a Packed Record Structure

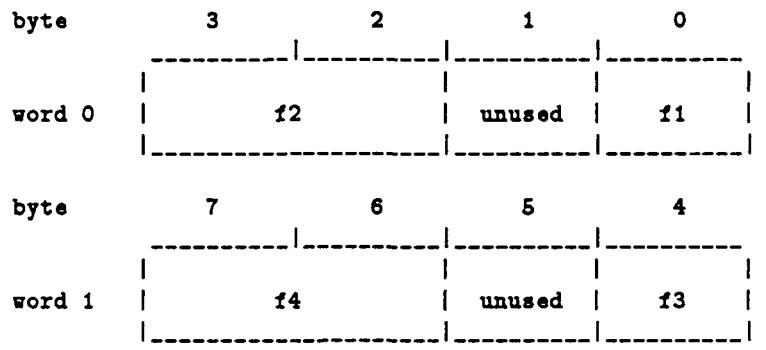

Figure 4. Pragma Pack Ignored

word unused. The fourth field will then use the first 10 bits of the next word. See Figure 3.

In the above example, the amount of memory used is the same as when it is unpacked. In such a situation, note that pragma *Pack* does not take effect, and the memory is allocated according to default memory allocation algorithm to maximize efficiency. See Figure 4 for an illustration.

Structure Objects Must Start on Byte Boundaries. Structure types always start on byte boundaries, even if the structure can be represented in fewer than 8 bits. If an array of records is declared as in the code segment below, each array element is one byte, so the entire array uses 32 bits, regardless of whether it is packed. See Figure 5 for an illustration.

```
type int4 is range 0..2**4-1;
for int4'size use 4;          -- INT4 is unsigned 4-bit integer

type sample_rec2 is
  record
    f1 : int4;        -- 4  bit field
  end record;

type sample_array is array(1..4) of sample_rec2;
```

Structures Are Aligned to Maximum Alignment of Their Components Structure types (record and arrays) are always aligned to the maximum alignment requirement of their individual components. This is because the compiler

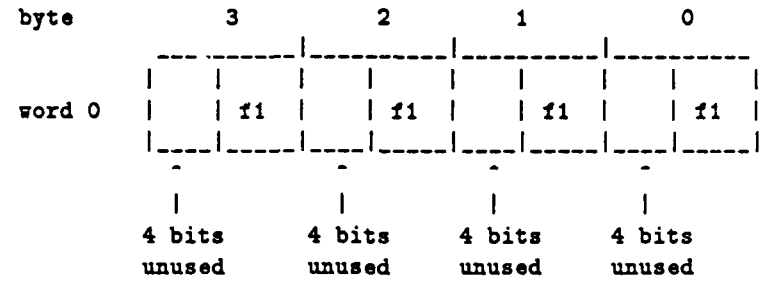

Figure 5. Memory Allocation for an Array of Records

must always guarantee that the first byte of every structure is at a byte meeting the alignment requirement of the largest field within the structure. (The compiler assures that each offset within a record is correct for the alignment requirement of each field. When the record is allocated in memory it must be properly aligned so that each of its component fields are properly aligned.)

Structures Are Sized to a Multiple of Their Alignment. Structure types (record and arrays) are always sized to a multiple of their alignment requirement so that when arrays of structures are declared each element of the array is guaranteed to be properly aligned.

For example, if a record is declared with two fields, a (32-bit) integer and a byte, then the total size of the record will be 64-bits, not the expected 40-bits. This is because the (default) alignment requirement for integers is 4-bytes. The compiler will (by default) make the alignment requirement for the record 4-bytes also. Since the alignment requirement for the record is 4-bytes its size must be a multiple of 4-bytes, so it is rounded up to 64-bits.

10.3 Address Clauses

Address clauses are implemented, but only for variables. Address clauses for local variables using dynamic values are now implemented properly. Although the LRM explicitly states that address clauses should not be used to overlay objects, the use of a dynamic address can facilitate such a feature. Address clauses are currently not implemented for subprograms, packages, tasks, entries, constant objects, or statement labels.

11 Interface to Other Languages

Pragma interface allows Ada programs to interface with (i.e., call) subprograms written in another language (e.g., assembly, C), and pragma export allows programs written in another language to interface with programs written in Ada. The accepted languages are: **Builtin**, **Ada**, **C** and **Assembly**. The aliases **Assembler** and **ASM** can also be used instead of **Assembly**. The langauge **Builtin** should be used with care–It is used by ICC for internally handled operators.

12 Unchecked Type Conversion

The generic function **Unchecked_conversion** is implemented. In general, unchecked_conversion can be used when the underlying representation of values is similar.
Acceptable conversions are:

- Conversion of scalars. Unchecked_conversion can be used to change the type of scalar values without restriction. In most circumstances the unchecked conversion produces no additional code.
- Conversion of constrained structures. Constrained arrays and records are represented as continguous areas of memory, and hence can be converted using unchecked_conversion.
- Conversion of scalars to constrained structures. Scalar objects may be converted to constrained structures with no additional overhead. If a scalar value is converted to a structure an aggregate is first built to hold the scalar value and its address is used as the address of the resulting structure.
- Conversion from unconstrained arrays. Since it is not possible to declare unconstrained array objects in Ada, conversion from unconstrained array types poses no difficulties. The rules described above are used.

Although the Ada compiler does *NOT* produce errors for the following unchecked conversions, they should be avoided since their results are not obvious:

- Conversion from discriminant records. Conversion from discriminant records can cause unpredictable behavior because of underlying representation changes. The unchecked_conversion will use the same rules as described above for performing the copy, however the results of this operation may not be what the user desires, since ICC Ada does *not* place arrays constrained by the discriminant in-line with the other fields in a discriminant record. In place of the array only a pointer is used and the array is allocated dynamically off of the (compiler maintained) heap.

- Conversion to or from pointers to unconstrained arrays. Unconstrained array pointers are implemented as special dope-vectors in ICC Ada. Conversion to or from these dope-vector is not recommended.

- Conversion to or from any type or object declared in a generic. Generics can cause hidden representation changes. Unchecked_conversions of any object or type declared in a generic should be avoided.

ICC Ada does not require that the sizes of the parameters to an unchecked_conversion be identical. The following rules are used when the source and target sizes differ:

- If the target type is constrained, then unchecked_conversion uses the size of the *target* type to determine the number of bytes to copy. The size of the *target* type (in bytes) is determined by the Ada frontend and exactly that many bytes are copied from the source address to the target address. This can cause problems (e.g. memory faults) when the source object is smaller than the target object. For example, using unchecked_conversion to convert a character into an integer will cause 4 bytes to be copied starting from the address of the character. The first byte copied will be the value of the character, but the values of the remaining three bytes cannot be predicted since they depend on values of variables or fields immediately after the character in memory. If the source object is larger than the target object then only the bytes that will fit in the target object are copied from the source starting at the address of the first byte of the source.

- If the target type is an unconstrained discriminant record, then unchecked_conversion uses the largest size possible for the record. Note that since ICC Ada does *not* place arrays constrained by discriminants in-line with the rest of the other fields in a record, this size may not be what the user expects.

13 Unchecked Storage Deallocation

Unchecked_deallocation is currently not implemented. For compatibility purposes unchecked_deallocation *can* be instantiated in ICC Ada programs, however calls will have no effect.

14 Machine Code Insertion

Machine code insertion is implemented. The ICC Ada compiler recognizes any record aggregate appearing in a sequence of statements as a machine code insertion. The interpretation of the code insertion is code generator dependent. The package MACHINE_CODE is currently not provided.

15 Main Programs

Main programs may be procedures or functions and may have any number of parameters. Parameter and function return types can be either discrete types (including enumerations) or unconstrained arrays. Parameters may also include default values. If the main program is a function, then upon exit the returned value will be printed on the user's screen. If the program is invoked with the wrong number of parameters a usage error message is printed and execution is aborted. If an illegal value is passed to a parameter then CONSTRAINT_ERROR is raised.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	96
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	(1..253 => 'A', 254 => '1')
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	(1..253 => 'A', 254 => '2')
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	(1..126 => 'A', 127 => '3', 128..254 => 'A')

TEST PARAMETERS

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	(1..126 => 'A', 127 => '4', 128..254 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..251 => '0', 252..254 => "298")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..249 => '0', 250..254 => "690.0")
$BIG_STRING1 A string literal which when catenated with $BIG_STRING2 yields the image of $BIG_ID1.	(1 => '"', 2..128 => 'A', 129 => '"')
$BIG_STRING2 A string literal which when catenated to the end of $BIG_STRING1 yields the image of $BIG_ID1.	(1 => '"', 2..127 => 'A', 128 => '1', 129 => '"')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..234 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	1048576
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	M68020
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	0.000000000465661287307739257812 5
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	524287.5
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10000000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	254
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	/NODIRECTORY/FILENAME
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/NODIRECTORY/THIS-FILE-NAME-IS-ILLEGAL
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648

TEST PARAMETERS

Name and Meaning	Value
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-524287.5
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10000000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	254
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be $MAX_IN_LEN long.	(1..2 => "2:", 3..251 => '0', 252..254 => "11:")

Name and Meaning	Value
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be $MAX_IN_LEN long.	(1..3 => "16:", 4..250 => '0', 251..254 => "F.E:")
$MAX_STRING_LITERAL A string literal of size $MAX_IN_LEN, including the quote characters.	(1 => '"', 2..253 => 'A', 254 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	TINY_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	M68000,M68010,M68020,M68030
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	1048576

TEST PARAMETERS

Name and Meaning	Value
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	M68010
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	(1.0/60.0)

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C: This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this text that must appear at the top of the page.

b. A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A: This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B: This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D: This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84M..N, and CD50110 (5 tests): These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86, 96, and 58, respectively).

i. CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B: This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C..D, and ED7006C..D (5 tests): These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).

q. CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, and 136).

s. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER OPTIONS AS SUPPLIED BY IRVINE

Compiler: ICC Ada, 6.0.0

ACVC Version: 1.10

The following is a list of ALL of the available compiler options:

```
ICC Flags
   -arguments -args              Display ICC arguments
   -asm                          Stop at the assembly file
   -asm_ext <arg>                Set assembly file extension
   -asm_flag -asmf <arg>         Explicitly add flag for assembler
   -asm_name <arg>               Use <arg> as the assembler
   -c                            Stop at the C file
   -cc_flag -ccf <arg>           Explicitly add flag for C compiler
   -cc_name <arg>                Use <arg> as the C compiler
   -coptimize -copt              Invoke C optimizer
   -debugger -debug              Compile code for Ada debugger
   -display -disp                Display exec calls
   -duo                          Use new 'duo' frontend/linker
   -exe                          Link non-Ada program
   -exe_ext <arg>                Set executable file extension
   -help                         List available flags
   -helpall -all                 List ALL available flags
   -hide                         Suppress naming ICC subprocesses
   -ignore_cfg -no_cfg -ncfg     Ignore config flags
   -ignore_env -no_env -nenv     Ignore environment flags
   -int                          Stop at the iform file
   -keep_temps -keep <arg>       Save files with extensions in <arg>
   -lib_ext <arg>                Set library file extension
   -library -lib <arg>           Explicitly set compilation library
   -list_merge -merge            Invoke ICC list merger
   -loader_name <arg>            Use <arg> as the loader
   -mrg                          Stop at the list merge file
   -mrg_ext <arg>                Set list merge file extension
   -no_temp                      Use local directory for intermediate files
   -no_unique                    Use real file name for intermediate files
   -normal -norm -n              Compile with normal messages
   -obj                          Stop at the object file [default]
   -obj_ext <arg>                Set object file extension
   -objlib <arg>                 Install the object file in library <arg>
   -objlib_flag -objlibf <arg>   Explicitly add flag for object librarian
   -objlib_name <arg>            Use <arg> as the object librarian
   -odebugger -odebug            Compile code for old Ada debugger
   -optimize -opt                Invoke ICC optimizer
   -postloader_flag -pstldf <arg>Explicitly add post-flag for loader
   -preloader_flag -preldf <arg> Explicitly add pre-flag for loader
   -quiet -q                     Compile quietly
   -rug <arg>                    Use <arg> as the rug name for binaries
   -runtime -run <arg>           Explicitly set runtime directory
   -save_pas2                    Save temporary files generated by pas2
   -save_temps -save             Save all intermediate files
   -show_only -show              Show only, don't exec
   -symbols -syms                Show current value of ICCCMD symbols
   -syntax_only -syntax          Syntax check only
   -temp -tmp <arg>              Use <arg> as a temporary directory
   -verbose -v                   Compile with verbose messages

Ada Flags
   -advise                       Suppress advisory warnings
   -cross_ref -cross             Generate cross-reference file (.xrf)
   -elaboration_check -elab      Generate ELABORATION checking
   -exception_info -ex           Suppress EXCEPTION_INFO
   -listing                      Generate list file (.lst)
   -no_checks -no                Suppress ALL_CHECKS
   -no_wrap                      Suppress auto-wrapping error messages
   -no_zero                      Apply pragma NO_ZERO to all records
   -preprocess -pre              Generate commented preprocess file (.app)
   -rate                         Rate code efficiency
   -stack_check                  Generate stack checking code
   -trim                         Generate trimmed preprocess file (.app)
   -warning -warn -w             Suppress warnings

ICC Code Generator Flags
   -68881                        Generate inline 68881 code
   -const_in_code                Place constant aggregates in CODE segment
   -dbx -cdb -xdb                Generate host debugger information
   -extended_list -extend -exl   Generate extended code output
   -fpa                          Generate inline FPA code
   -gprofile -gprof              Generate runtime gprofiling
   -loc_info                     Generate extended local information
   -names                        Generate namelist file
   -numeric_check -numchk        Generate overflow detection code
   -probe_stack -probe           Generate stack probes
   -profile -prof                Generate runtime profiling
   -real                         Use realnames
   -relative                     Use relative branches (BSD only)
   -static                       Static flag
```

ICC Prelinker Flags

-complink <arg>	Ada compile and link <arg> into one file
-force_link	Force link, even if dependency errors
-link -l <arg>	Ada link compilation unit <arg>
-no_trap	Don't establish trap handler for numeric faults
-output -out -o <arg>	Make executable file <arg>
-stock <arg>	Allocate <arg> 8KB pages for user stack
-vms_debugger -vms_debug	Link with VMS Debugger